Programming Languages

2nd edition

Tucker and Noonan

Chapter 7
Semantics

Surely all this is not without meaning.

Ishmael, Moby Dick by Herman Melville

Contents

7.1 Motivation

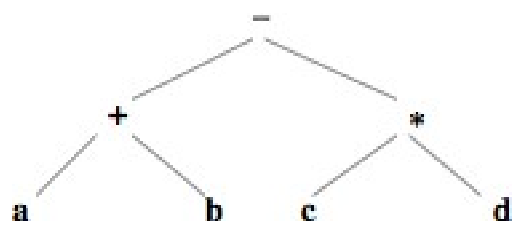
- To provide an authoritative definition of the meaning of all language constructs for:

1. Programmers
2. Compiler writers
3. Standards developers

A programming language is complete only when its syntax, type system, and semantics are well-defined.

- Semantics is a precise definition of the meaning of a syntactically and type-wise correct program.
- Ideas of meaning:
  - *The meaning attached by compiling using compiler C and executing using machine M. Ex: Fortran on IBM 709.*
  - *Axiomatize statements -- Chapter 12*
  - *Statements as state transforming functions*
- This chapter uses an informal, operational model.

7.2 Expression Semantics



- (a + b) - (c * d)
- Polish Prefix: - + a b * c d
- Polish Postfix: a b + c d * -
- Cambridge Polish: (- (+ a b) (* c d))

Infix uses associativity and precedence to disambiguate.

Associativity of Operators

Language	+ - * /	Unary -	**	== != < ...
C-like	L	R		L
Ada	L	non	non	non
Fortran	L	R	R	L

Meaning of: `a < b < c`

Precedence of Operators

Operators	C-like	Ada	Fortran
Unary -	7	3	3
**		5	5
* /	6	4	4
+ -	5	3	3
== !=	4	2	2
< <= ...	3	2	2
not	7	2	2

Short Circuit Evaluation

a and b evaluated as:

    if a then b else false

a or b evaluated as:

    if a then true else b

Example

```
Node p = head;
while (p != null && p.info != key)
        p = p.next;
if (p == null) // not in list
        ...
else // found it
        ...
```

Versus

```
boolean found = false;
while (p != null && ! found) {
        if (p.info == key)
                found = true;
        else
                p = p.next;
}
```

Side Effect

A change to any non-local variable or I/O.

What is the value of:

```
i = 2; b = 2; c = 5;
a = b * i++ + c * i;
```

7.3 Program State

The state of a program is the collection of all active objects and their current values.

Maps:

1. The pairing of active objects with specific memory locations,
2. and the pairing of active memory locations with their current values.

The current statement (portion of an abstract syntax tree) to be executed in a program is interpreted relative to the current state.

The individual steps that occur during a program run can be viewed as a series of state transformations.

For the purposes of this chapter, use only a map from a variable to its value; like a debugger watch window, tied to a particular statement.

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
          i = i + 1;
          f = f * i;
  }
}
```

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
          i = i + 1;
          f = f * i;
  }
}
```

n	i	f
undef	undef	undef
3	undef	undef

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                  i = i + 1;
8                  f = f * i;
9      }
10   }
```

n	i	f
3	undef	undef
3	1	undef

// compute the factorial of n		n	i	f
1	void main ( ) {			
2	int n, i, f;			
3	n = 3;			
4	i = 1;			
5	f = 1;	3	1	undef
6	while (i < n) {	3	1	1
7	i = i + 1;			
8	f = f * i;			
9	}			
10	}			

```
// compute the factorial of n
1     void main ( ) {
2      int n, i, f;
3       n = 3;
4       i = 1;
5       f = 1;
6       while (i < n) {
7                   i = i + 1;
8                   f = f * i;
9       }
10    }
```

n	i	f
3	1	1
3	1	1

```
// compute the factorial of n
1     void main ( ) {
2      int n, i, f;
3       n = 3;
4       i = 1;
5       f = 1;
6       while (i < n) {
7                   i = i + 1;
8                   f = f * i;
9       }
10    }
```

n	i	f
3	1	1
3	2	1

	// compute the factorial of n	n	i	f
1	void main ( ) {			
2	int n, i, f;			
3	n = 3;			
4	i = 1;			
5	f = 1;			
6	while (i < n) {	3	2	2
7	i = i + 1;			
8	f = f * i;	3	2	1
9	}			
10	}			

// compute the factorial of n		n	i	f
1	void main ( ) {			
2	int n, i, f;			
3	n = 3;			
4	i = 1;			
5	f = 1;			
6	while (i < n) {	3	2	2
7	i = i + 1;	3	2	2
8	f = f * i;	3	3	2
9	}	3	3	6
10	}			

```
// compute the factorial of n                n       i       f
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {                       3       3       6
7              i = i + 1;
8              f = f * i;
9      }
10   }
                                             3       3       6
```

7.4 Assignment Semantics

Issues

- Multiple assignment
- Assignment statement vs. expression
- Copy vs. reference semantics

Multiple Assignment

Example:

```
a = b = c = 0;
```

Sets all 3 variables to zero.

Problems???

Assignment Statement vs. Expression

- In most languages, assignment is a statement; cannot appear in an expression.
- In C-like languages, assignment is an expression.
  - *Example:* if (a = 0) ... // an error
  - while (*p++ = *q++) ; // strcpy
  - while (ch = getc(fp)) ... // ???
  - while (p = p->next) ... // ???

Copy vs. Reference Semantics

- Copy: a = b;
  - a, b *have same value.*
  - *Changes to either have no effect on other.*
  - *Used in imperative languages.*
- Reference
  - a, b *point to the same object.*
  - *A change in object state affects both*
  - *Used by many object-oriented languages.*

```
public void add (Object word, Object number) {
    Vector set = (Vector) dict.get(word);
    if (set == null) {  // not in Concordance
        set = new Vector( );
        dict.put(word, set);
    }
    if (allowDupl || !set.contains(number))
        set.addElement(number);
}
```

7.5 Control Flow Semantics

To be complete, an imperative language needs:

- Statement sequencing
- Conditional statement
- Looping statement

Sequence

s1 s2

Semantics: in the absence of a branch:

- First execute s1
- Then execute s2
- Output state of s1 is the input state of s2

Conditional

$\mathit{IfStatement} \rightarrow$ if ($\mathit{Expresion}$) $\mathit{Statement}$

[else $\mathit{Statement}$]

Example:

```
if (a > b)
    z = a;
else
    z = b;
```

If the test expression is true,

then the output state of the conditional is the output state of the then branch,

else the output state of the conditional is the output state of the else branch.

Loops

WhileStatement → while (*Expression*) *Statement*

The expression is evaluated.

If it is true, first the statement is executed,

and then the loop is executed again.

Otherwise the loop terminates.

7.6 Input/Output Semantics

- Binding: open, close
- Access: sequential vs. random
- Stream vs. fixed length records
- Character vs. binary
- Format

Standard Files

- Unix: stdin, stdout, stderr
- C: stdin, stdout, stderr
- C++: cin, cout, cerr
- Java: System.in, System.out, System.err

Input/Output Streams

- Fortran

```
integer :: i, a(8)
write(8,*) “Enter 8 integers: “
read(*,*) a
write(*,*) a
```

- Java
  - file, pipe, memory, url
  - filter
  - reader, writer

Formats

- C
  - *Codes: d, e, f, c, s (decimal. float, float, char, string)*
  - *Specifier: % opt-width code*
  - *Ex: %s %5d %20s %8.2f*
- Fortran
  - *Codes: i, f, a (integer, float, string)*
  - *Specifier: op-repeat code width*
  - *Ex: 8i4, f8.2, a20*

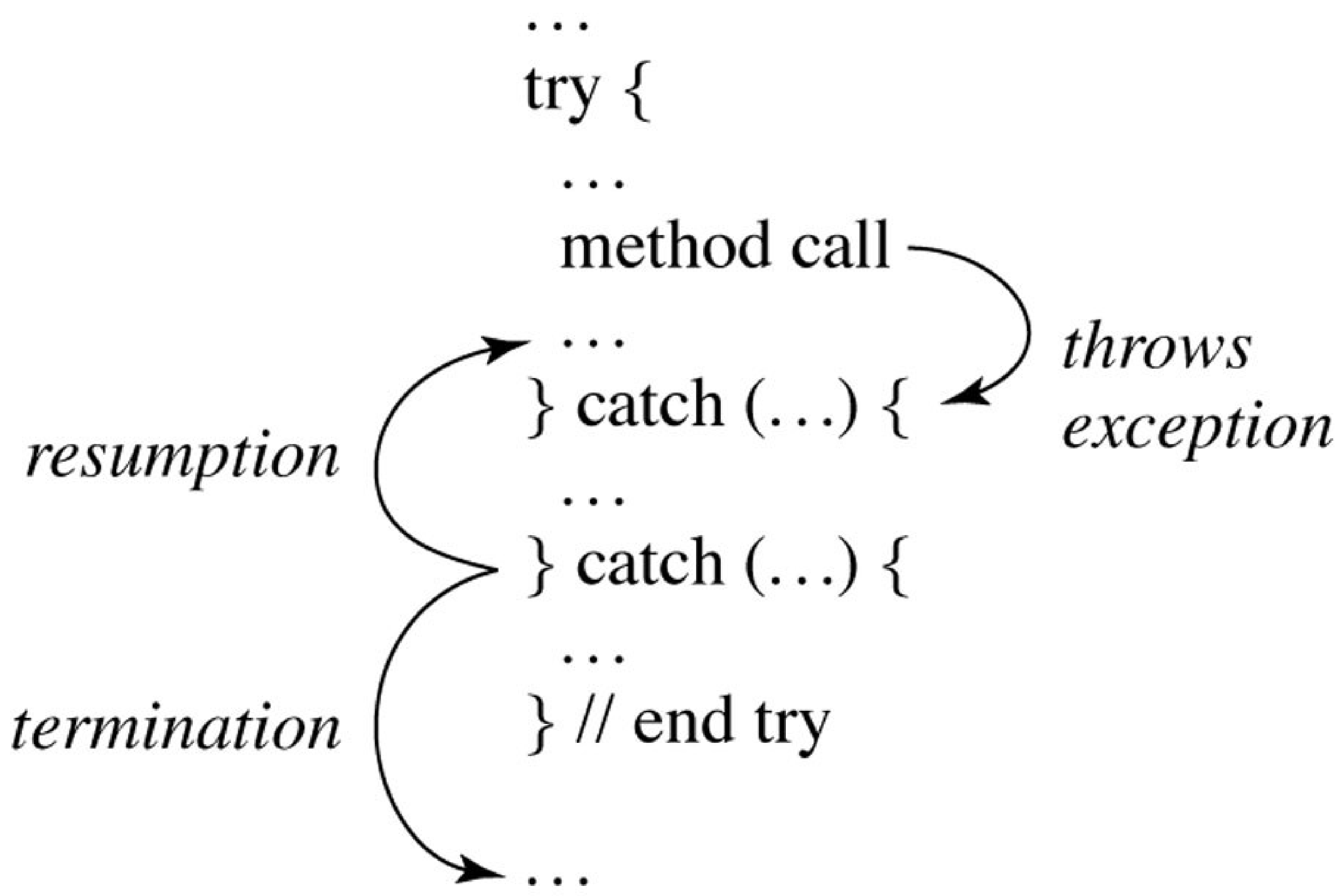
Purpose

- Simplify programming
- Make applications more *robust*.
- What does *robust* mean?

```
(* Pascal - what can go wrong *)
reset(file, name);
  (* open *)
sum := 0.0;
count := 0;
while (not eof(file)) do begin
   read(file, number);
   sum := sum + number;
   count := count + 1;
end;
ave := sum / count;
```

Exception Handling

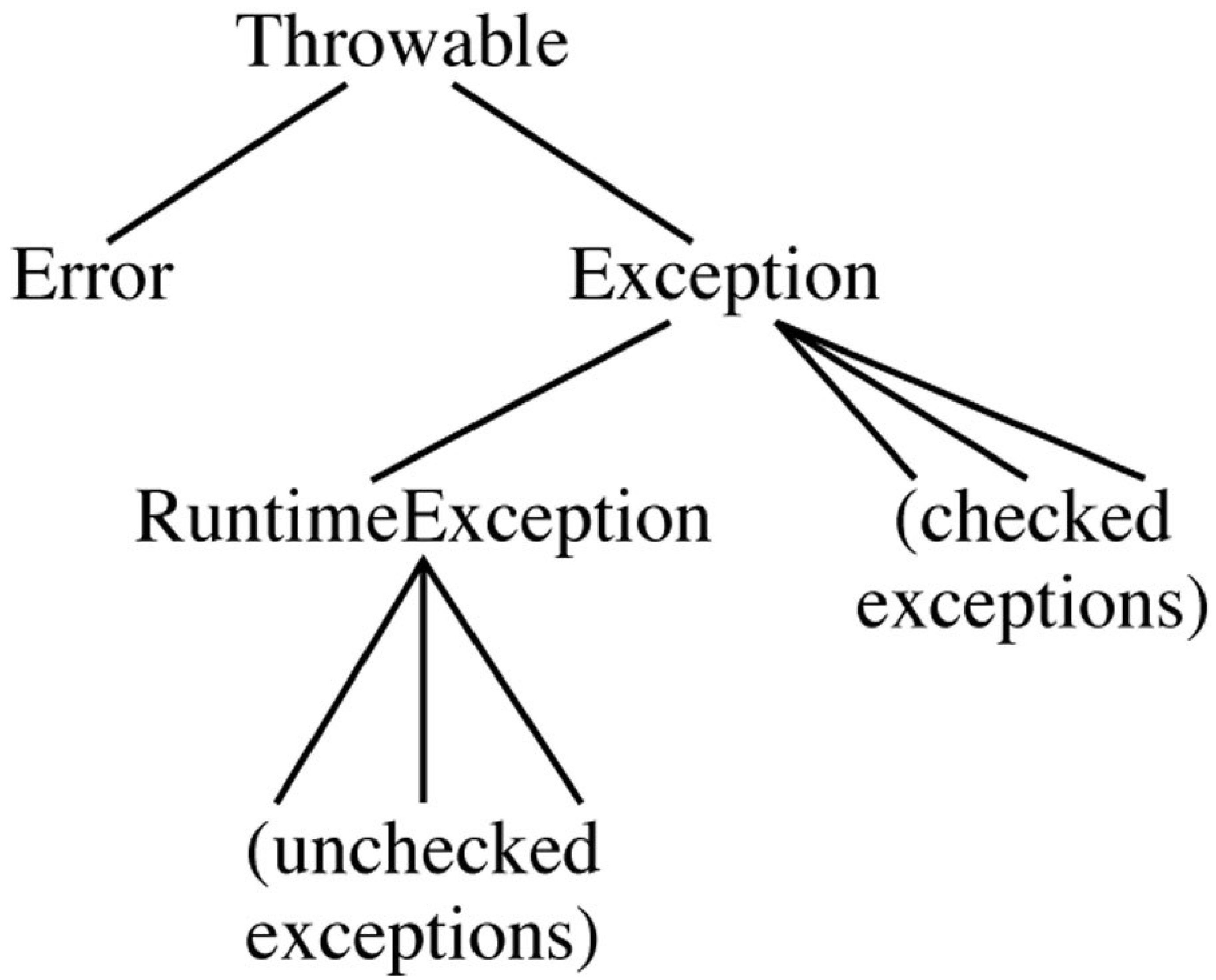
Figure 7.9



```
#include <iostream.h>
int main () {
  char A[10];
  cin >> n;
  try {
    for (int i=0; i<n; i++){
      if (i>9) throw "array index error";
      A[i]=getchar();
    }
  }
  catch (char* s)
  { cout << "Exception: " << s << endl; }
  return 0;
}
```

Java Exception Type Hierarchy

Figure 7.10



Creating a New Exception Class

```
class StackUnderflowException extends Exception {
   public StackUnderflowException() { super(); }
   public StackUnderflowException(String s){  super(s);}
}
```

Missing Argument Exception

```
public static void main(String[] arg) {
   try {
   if (arg.length < 1)  {
   System.err.println("Missing argument.");
   displayUsage( );
   System.exit(1);
   }
   process(new BufferedReader(new FileReader(arg[0])));
 } catch (FileNotFoundException e) {
   System.err.println("Cannot open file: " + arg[0]);
   System.exit(1);
 }
}
```

Invalid Input Exception

Figure 7.12

```
while (true) {
  try {
    System.out.print ( "Enter number : ");
    number = Integer.parseInt(in.readLine ( ));
    break;
  } catch (NumberFormatException e) {
    System.out.println ( "Invalid number, please reenter.");
  } catch (IOException e) {
    System.out.println("Input error occurred, please reenter.");
  } // try
} // while
```

StackUnderflowException Class

Figure 7.13

```
class StackUnderflowException extends Exception {
  public StackUnderflowException( ) { super(); }
  public StackUnderflowException(String s){ super(s);}
}
```

Throwing an Exception
Figure 7.13

```
class Stack {
  int stack[];
  int top = 0;
  ...

  public int pop() throws StackUnderflowException {
    if (top <= 0)
        throw new StackUnderflowException("pop on empty stack");
    return stack[--top];
  }
  ...
}
```

AssertException Class

```
class AssertException extends RuntimeException {
    public AssertException( ) { super( ); }

    public AssertException(String s) { super(s); }
}
```

Assert Class

Figure 7.15

```
class Assert {
    static public final boolean ON = true;
    static public void isTrue(boolean b) {
        if (!b) { throw new AssertException(“Assertion failed”); }
    }

    static public void shouldNeverReachHere() {
        throw new AssertException(“Should never reach here”);
    }
}
```

Using Asserts

```
class Stack {
  int stack[];
  int top = 0;
  . . .

  public boolean empty() { return top <= 0; }

  public int pop() {
    Assert.isTrue(!empty());
    return stack[--top];
  }
  . . .
}
```